CMSC 216
Introduction to Computer Systems
Lecture 10
Pointers, cont. and Make

Chapter 11, Reek

Dynamic Memory Allocation (cont.)

Pointer aliases

- We can have two pointers pointing to the same address
- Remember freeing the space doesn't change the value of the pointer or the value of the space pointed at

```
int *p, *q;
p = malloc(sizeof(int));
*p = 99;
q = p;   /*values of p and q and *p and *q*/
free(p);
p = NULL;
*q = 42;
```

- `q` is called a **dangling pointer**

Common errors

- Dereferencing pointers to freed space (directly or indirectly)
- Forgetting to check the return value from **`malloc()`** for **`NULL`**
- Not initializing the memory **`malloc()`** returns
- Not allocating enough space:

```
char string[] = "Inconceivable!";
char *p = malloc(strlen(string));
strcpy(p, string);  /* Oops... */
```

Common errors, cont.

- Attempting to free non-heap memory:

```
int i, *p;
p = &i;
free(p);  /* ??? Undefined operation */
```

- Freeing something that's not the **beginning** of a dynamically allocated block:

```
int *p = calloc(10, sizeof(int));
free(p + 3);
```

Common errors, cont.

- Dereferencing a random address (garbage pointer)

```
int *p;
*p = 42;
```

- Dereferencing a **NULL** pointer

```
int *p;
p = NULL;
*p = 23;
```

- Dereferencing a dangling pointer

```
int *p = malloc(...);
free(p);
...
*p = 27;
```

Common errors, cont.

- Referring to data past the end of allocated space
  - Same as statically allocated arrays

- Using freed memory

```
List_node *ptr =
            malloc(sizeof(List_node));
...
free(ptr);
...
ptr = ptr->next;  /* Uh oh... */
```

Common errors, cont.

- What happens to the first **`Stack`** object in Java?

```
Stack s = new Stack();
...
s = new Stack();
```

- Losing a pointer to dynamically allocated memory causes **memory leaks**
  - not an immediately fatal error
  - can just be a waste of resources, but left to run for a while, it can keep your program from being able to do anything

Reallocation

- If you need more space than you originally allocated, you could handle this yourself:
  - allocate array twice as large
  - copy everything over
  - free original array
- But, there's a way to do this without nearly as much work on your part

Reallocation, cont.

```
void *realloc(void *p, size_t new_size);
```

- checks current size of the block **`p`** points to
  - if original size >= **`new_size`**, can perform reallocation in place, and returns **`p`**
  - if original size < **`new_size`**, new block of size **`new_size`** is allocated
    - if allocation fails, returns **`NULL`**
    - otherwise, copies items from **`p`** over, free **`p`**, and returns pointer to new block

Use of `realloc()`

- Often, you'll see this:

```
ptr = realloc(ptr, size * 2);
```

- But what happens if the allocation fails?
- You can only do this if you know for certain you're going to exit the program on an allocation failure

Operating on blocks of memory

- Much as we have functions like **`strcpy()`** that operate on strings of characters, we can copy whole blocks of memory (or strings of bytes, if you like)

```
void *memcpy(void *dst, void *src, size_t n);
void *memmove(void *dst, void *src, size_t n);
```

- Both copy **`n`** bytes from the memory starting at **`src`** to the memory starting at **`dst`**, and return the **`dst`** pointer
- **`memcpy()`** cannot be used on overlapping arrays, though; but it's faster than **`memmove()`**
- Both prototypes contained in **`string.h`**

A possible `memcpy()` implementation

```
#include <stdio.h>
/* not #including <string.h> because of conflict */

void *memcpy(void *dst, void *src, size_t n) {
  char *dp = dst;
  char *sp = src;
  while (dp - (char *) dst < n)
    *dp++ = *sp++;
  return dst;
}

int main() {
  int i, j = 5;
  memcpy(&i, &j, sizeof(int));
  printf("%d\n", i);
  return 0;
}
```

Chapter 12

Using Structures and Pointers

Linked lists

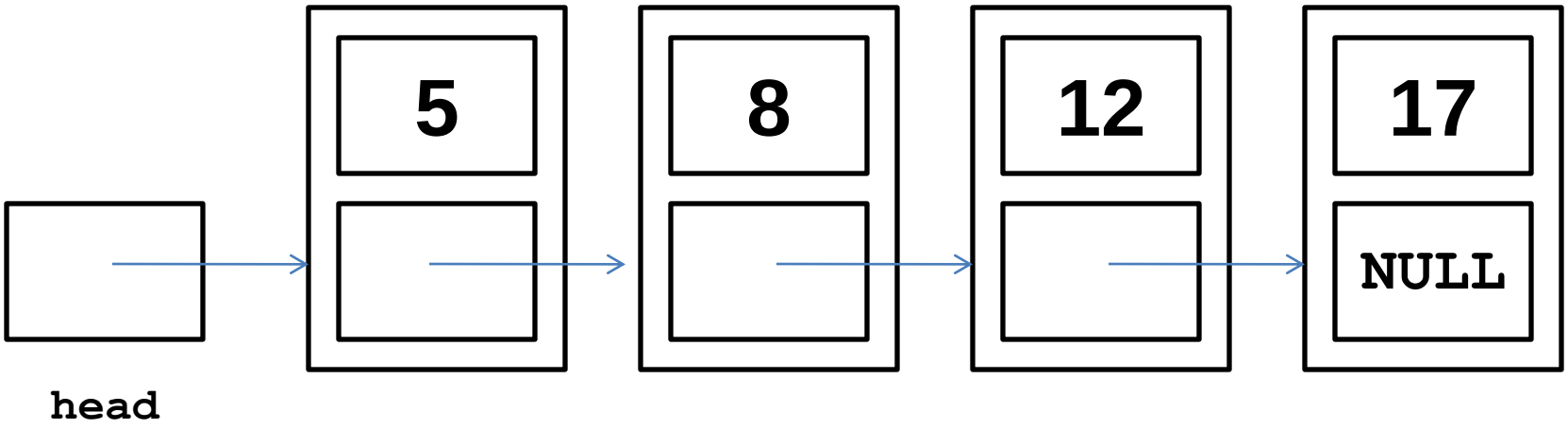
- Much like Java's objects with references to objects of that class, C structures can have pointers to structures of the same type

```
typedef struct node {
  int val;
  struct node *next;
} Node;

Node *head;
```

- Both **`next`** and **`head`** are of the same type

Linked list in memory

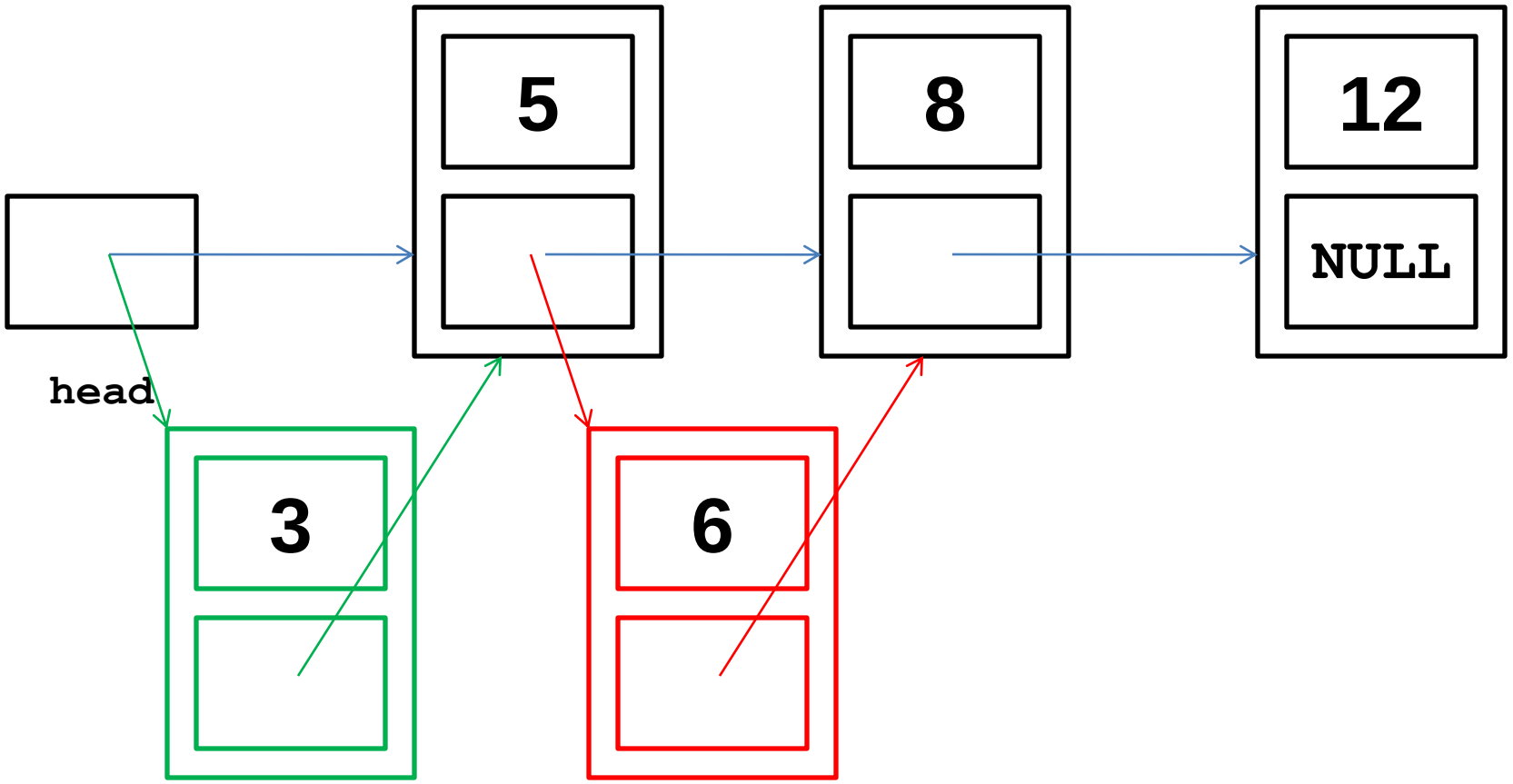


Searching through a linked list

```
Node *find(Node *start, int target) {
  while (start != NULL) {
    if (start->val == target)
      return start;
    start = start->next;
  }
  return NULL;
}

int find(Node *start, int target) {
  while (start != NULL && start->val != target)
    start = start->next;
  return start != NULL;
}
```

Inserting into an ordered linked list



Tracing insertion

```
int insert(Node *head, int new_value) {
  Node *current = head, *prev = NULL, *new_item;

  while (current != NULL && current->val < new_value) {
    prev = current;
    current = current->next;
  }
  new_item = malloc(sizeof(*new_item));
  if (new_item == NULL)
    return 0;
  new_item->val = new_value;
  new_item->next = current;
  if (prev == NULL)
    head = new_item;
  else
    prev->next = new_item;
  return 1;
}
```

Tracing insertion - fixed

```
int insert(Node **head, int new_value) {
  Node *current = *head, *prev = NULL, *new_item;

  while (current != NULL && current->val < new_value) {
    prev = current;
    current = current->next;
  }
  new_item = malloc(sizeof(*new_item));
  if (new_item == NULL)
    return 0;
  new_item->val = new_value;
  new_item->next = current;
  if (prev == NULL)
    *head = new_item;
  else
    prev->next = new_item;
  return 1;
}
```

Use of Dummy Node in Linked List

- Node with no data
- Simplifies insertions and deletions
  - head pointer will never change
- Let's take a look

About sizeof and arrays

- Be careful with the use of sizeof
- sizeof where array is declared provides the size of the array
- If you pass the array to function what sizeof returns??